



Rapport TER HLIN601
Licence Informatique Troisième année ,
effectué de Janvier à Avril 2015,
encadré par Abdelhak-Djamel SERIAI



Surveiller sa maison via son Smartphone sous Android

RAPPORT

Travail réalisé par l'équipe



Abdoulaye DIALLO
Quentin PHILIPPOT
Jihen FOURATI
Redoine EL-OUASTI

<https://sourceforge.net/projects/ter2015/>

Remerciements

Nous remercions notre référent, M. Seriari, pour ses conseils avisés et pour l'efficacité avec laquelle il nous a guidés tout au long du développement du projet. Nous tenons à remercier aussi toutes les personnes présentes lors de notre soutenance et plus particulièrement les membres du Jury.

À propos du logo de l'application, il s'agit en réalité d'un emprunt au séduisant logo de l'équipe de Hockey "Rapace".

Table des matières

1	Introduction	5
2	Contexte et Étude des systèmes existants	6
2.1	Contexte	6
2.2	Télésurveillance et vidéosurveillance	6
2.2.1	Présence de télésurveillance	6
2.2.2	Poste de surveillance	6
2.3	Application mobile	7
2.4	Système d'exploitation mobile et Android	7
2.4.1	L'apport des applications mobiles	7
2.5	Aperçu des systèmes existants	8
2.6	L'application Rapace	8
3	Gestion du projet, Analyse et conception	9
3.1	Cahier des charges	9
3.2	Architecture	9
3.3	Cas d'utilisation	10
3.3.1	Schéma	10
3.3.2	Détail	10
3.4	Diagramme de classes	11
3.5	Diagramme de séquence	13
3.6	Base de données	13
4	Réalisation et implémentation	15
4.1	Interface Homme Machine :	15
4.2	Communication client - serveur	16
4.2.1	Côté client Rapace	16
4.2.2	Côté serveur Apache	17
4.3	Mécanisme d'alerte :	19
4.4	Lecture streaming :	20

5	Intérêts de ce projet	21
5.1	L'enrichissement personnel	21
5.2	Des compétences techniques	21
6	Conclusion	22
7	Bibliographie	23
8	Annexe	24
8.1	Mode d'emploi	24
8.1.1	Réception de l'alerte	24
8.1.2	L'Authentification	25
8.1.3	Affichage des sites	26
8.1.4	Streaming	26
8.2	Complément sur les communications Client-Serveur	30
8.2.1	Pourquoi le package network ?	30
8.2.2	Les processus légers ou Thread	33
8.2.3	Accès à la base de données	34
8.2.4	Comment est encodée la réponse du serveur ?	34
8.3	Nativité d'une application	36
8.4	Envoyer un SMS à l'émulateur Android	37
8.5	Code source	37
8.5.1	Utilisateur	38
8.5.2	Site	39
8.5.3	MessageHTTP	41
8.5.4	MessagePOST	44
8.5.5	Serveur	46
8.5.6	SMSReceiver	49

1 Introduction

Étudiants en troisième année de licence informatique, notre projet consiste à développer une application pour smartphone permettant de surveiller un lieu à distance.

Par sa vocation à surveiller, notre travail s'apparente à la télésurveillance, plus précisément à la vidéo-surveillance. Nous verrons quelles techniques sont associées à la "*Télésurveillance*". Autre domaine en lien avec notre projet, celui de la programmation mobile, qui consiste en la programmation de logiciels pour appareils mobiles. En mariant ces deux domaines, nous tâcherons d'associer la performance de l'un à la mobilité de l'autre afin de proposer la meilleure expérience possible à l'utilisateur.

Pour répondre à ces objectifs, nous avons connu plusieurs phases (gestion de projet, analyse et conception, définition de l'architecture, implémentation) durant lesquelles notre organisation évolua. Collégiale au début (conception et apprentissage d'Android), notre organisation s'est ensuite structurée : chacun se spécialisa dans un domaine (IHM pour Redoine, noyau l'application cliente pour Abdoulaye et Jihen, serveur Rapace pour Quentin).

Ce rapport est écrit de sorte à initier le lecteur aux problématiques rencontrées au cours du projet, et d'apporter des explications concises aux solutions proposées. Nous approfondirons certains points en annexe.

Pour commencer nous présenterons le contexte dans lequel se place notre projet et l'étude des systèmes existants, lesquels guideront notre analyse durant la phase de conception. Nous détaillerons ensuite la réalisation et l'implémentation. Enfin, nous porterons un regard sur l'ensemble du projet et discuterons de son intérêt pour notre formation.

2 Contexte et Étude des systèmes existants

Cette section a pour but d'expliciter le contexte dans lequel notre projet s'est déroulé.

2.1 Contexte

2.2 Télésurveillance et vidéosurveillance

La télésurveillance désigne un ensemble de techniques utilisant des réseaux informatiques, dispositifs de capture et de traitement de l'information. Son but est d'assurer la surveillance d'un lieu (entre autre) à distance.

La vidéosurveillance est une branche active de la télésurveillance, reposant sur l'usage de caméras de surveillances. Il s'agit alors de garantir la protection d'un lieu par l'étude d'images en provenance de ce dernier.

2.2.1 Présence de télésurveillance

- L'évaluation du trafic routier (fréquentation, pollution, etc).
- Aide à la maintenance des machines (usure, disponibilité ressources, détection d'erreur, etc).
- Surveillance de lieux sensibles (DBA, centres commerciaux) et moins sensibles (résidences privées).

2.2.2 Poste de surveillance

La télésurveillance force un individu à utiliser un poste de surveillance afin de visualiser les informations récoltées. La vidéosurveillance d'un supermarché, par exemple, impose aux vigiles d'être dans une pièce contenant des moniteurs pour voir les images obtenues grâce aux caméras.

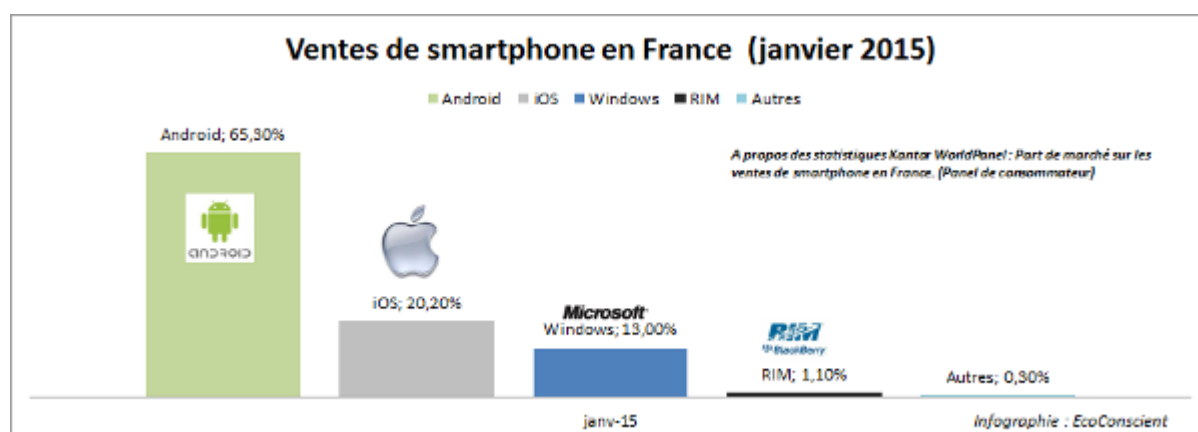
Dans le cas général, un utilisateur doit accéder à un poste de surveillance (moniteur, ordinateur, etc.) pour visualiser des informations.

2.3 Application mobile

Il s'agit d'un logiciel applicatif pour appareil mobile. L'avantage de ces applications est d'apporter une plus grande liberté à son utilisateur. Ce dernier pouvant accéder à ses applications favorites, où qu'il soit, simplement en allumant son appareil mobile.

2.4 Système d'exploitation mobile et Android

Il existe plusieurs systèmes d'exploitations pour les appareils mobiles (smartphones, tablettes, etc). ANDROID, IOS et WINDOWS PHONE dominent le marché.



Android se distingue des autres *OS* par la simplicité avec laquelle nous pouvons créer de nouvelles applications. Le choix de Java comme langage natif (voir annexe) libère les développeurs des contraintes liées à la gestion mémoire. De plus l'interprétation de code Java par la machine virtuelle ANDROID RUNTIME permet aux entreprises de créer rapidement des applications efficaces.

2.4.1 L'apport des applications mobiles

Ce qu'apportent les applications mobiles à la télésurveillance, c'est la capacité de transformer son appareil mobile en un véritable poste de surveillance, via des applications de télésurveillance.

2.5 Aperçu des systèmes existants

Il existe sur le marché plusieurs solutions afin de transformer son mobile en poste de surveillance :

- **iCamSpy** (Android) : transforme son pc ou sa webcam en caméra de surveillance. Visualisation depuis son mobile. Détection de mouvements, mécanisme d’alerte. (Version payante : enregistrement vidéo)
- **AtHome** (iOS) : transforme une webcam en caméra de surveillance. Visualisation depuis son mobile. (Version payante : limite de webcams augmentée, enregistrement vidéo).
- **PrynPocket** (iOS) : Enregistrement programmé. Visualisation depuis son mobile. Mécanisme d’alerte. Contrôle du matériel à distance (etc.)

Celles-ci exigent une préparation de la part de l’utilisateur, (manipulation et/ou installation de logiciel/serveur pour iCamSpy et AtHome, achat de matériels de l’entreprise pour PrynPocket).

Un second type d’application existe, ne nécessitant “aucune installation”, permettant uniquement de lire le flux vidéo à partir d’une url (adresse diffusant le flux). Exemple d’applications : **IPCamViewer**, **IPWebcam** ...

2.6 L’application Rapace

Nous proposons une application nommée **Rapace**, appartenant à la première famille d’applications présentées précédemment. Nous considérerons toutefois qu’un utilisateur peut connecter autant de caméras qu’il le souhaite, et ne ferons donc pas le hiatus “application gratuite/payante”.

3 Gestion du projet, Analyse et conception

Cette section expose notre analyse du problème et la conception de l'application Rapace.

3.1 Cahier des charges

L'application Rapace (client), devra implémenter les fonctionnalités suivantes :

- L'authentification d'un utilisateur enregistré.
- La présentation et l'accès sécurisé aux informations relatives à un SITE (nous nommerons ainsi un lieu placé sous surveillance) à un utilisateur autorisé.
- La lecture en streaming d'une vidéo.

D'autres fonctionnalités pourront venir se greffer par la suite :

- Mécanisme d'alerte avertissant en cas d'intrusion (simulation logicielle).
- Possibilité de contacter directement la police sans quitter l'application.
- Gérer le cas de fausses alertes depuis l'application Rapace.

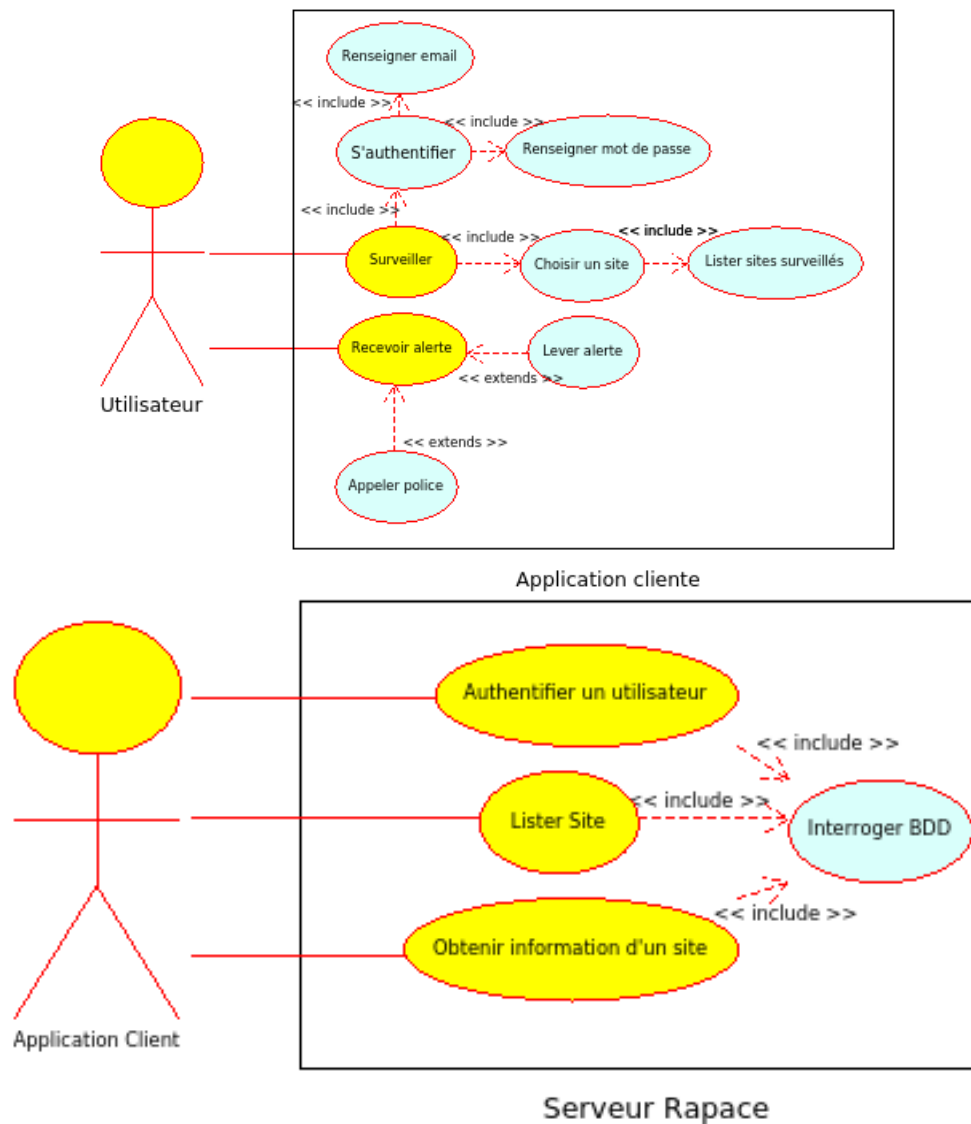
3.2 Architecture



L'architecture sera la suivante : l'application cliente communiquera avec le serveur afin d'interagir avec la base de données ou de récupérer un flux vidéo. Le serveur Rapace sera en réalité composé de deux serveurs : un serveur Apache pour les scripts PHP, et un serveur MySQL gérant nos données.

3.3 Cas d'utilisation

3.3.1 Schéma



3.3.2 Détail

L'authentification : Après enregistrement d'un utilisateur dans la base de données, ce dernier possède une adresse e-mail et un mot de passe qui régiront son authentification. Un message d'erreur pourra être affiché dans les cas suivants :

- Les données renseignées par le champ email ne correspondent pas à une adresse e-mail.
- Les informations fournies ne permettent pas l'authentification.

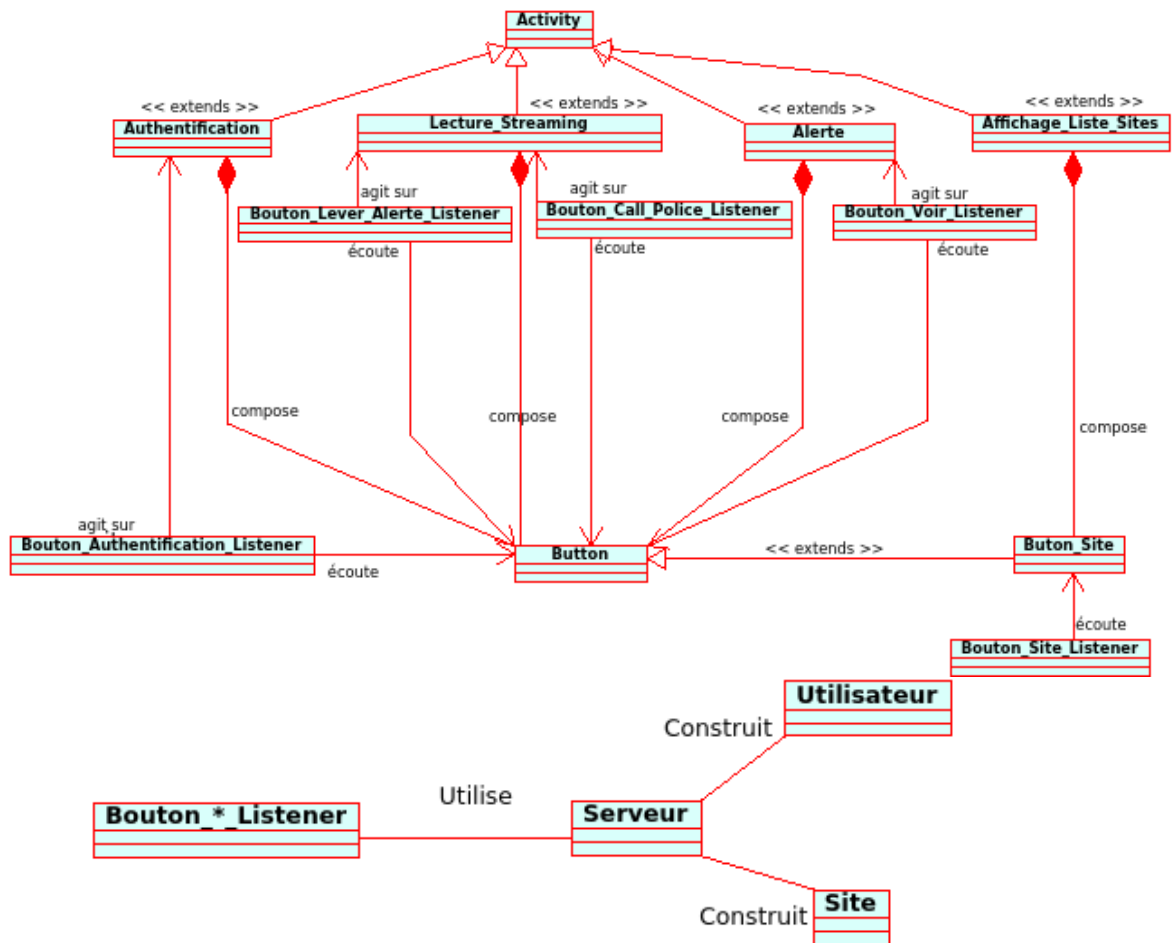
Affichage d'une liste de Sites : Cette activité permet la sélection d'un site parmi une liste de sites surveillés. L'IHM est dynamique, puisque l'affichage diffère selon le nombre de sites surveillés et leur état d'alerte (voir manuel en annexe). Chaque bouton correspond à un site. Une pression redirige l'utilisateur vers la vidéo du site sélectionné.

Surveiller ou Lecture d'un flux vidéo : L'activité lit un flux vidéo correspondant au site sélectionné. En cas d'alerte, deux boutons apparaissent sous la vidéo permettant soit d'appeler la police, soit de lever l'alerte.

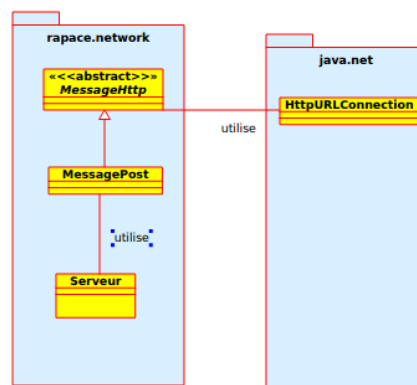
Réception d'une alerte : L'activité affiche une alerte. Pour des raisons de sécurité, on donnera peu de détails sur la nature d'une alerte (aucune information sur le site). Nous demandons à l'utilisateur de s'authentifier avant d'obtenir plus d'informations. Si l'utilisateur égare son téléphone et reçoit une alerte, aucune donnée ne peut être lue par un tiers.

3.4 Diagramme de classes

Notre conception utilise le patron MVC. On associe des contrôleurs à certains éléments graphiques comme suivant. Les classes *Bouton_*_Listener* sont des contrôleurs, *Bouton* ou *Bouton_Site* des éléments graphiques écoutés. Tous les contrôleurs (excepté *Bouton_Site_Listener*) sont en association avec l'activité contenant l'élément graphique). Les contrôleurs amorcent les transitions entre activités, et les interactions avec la classe métier (via la classe *Serveur*).

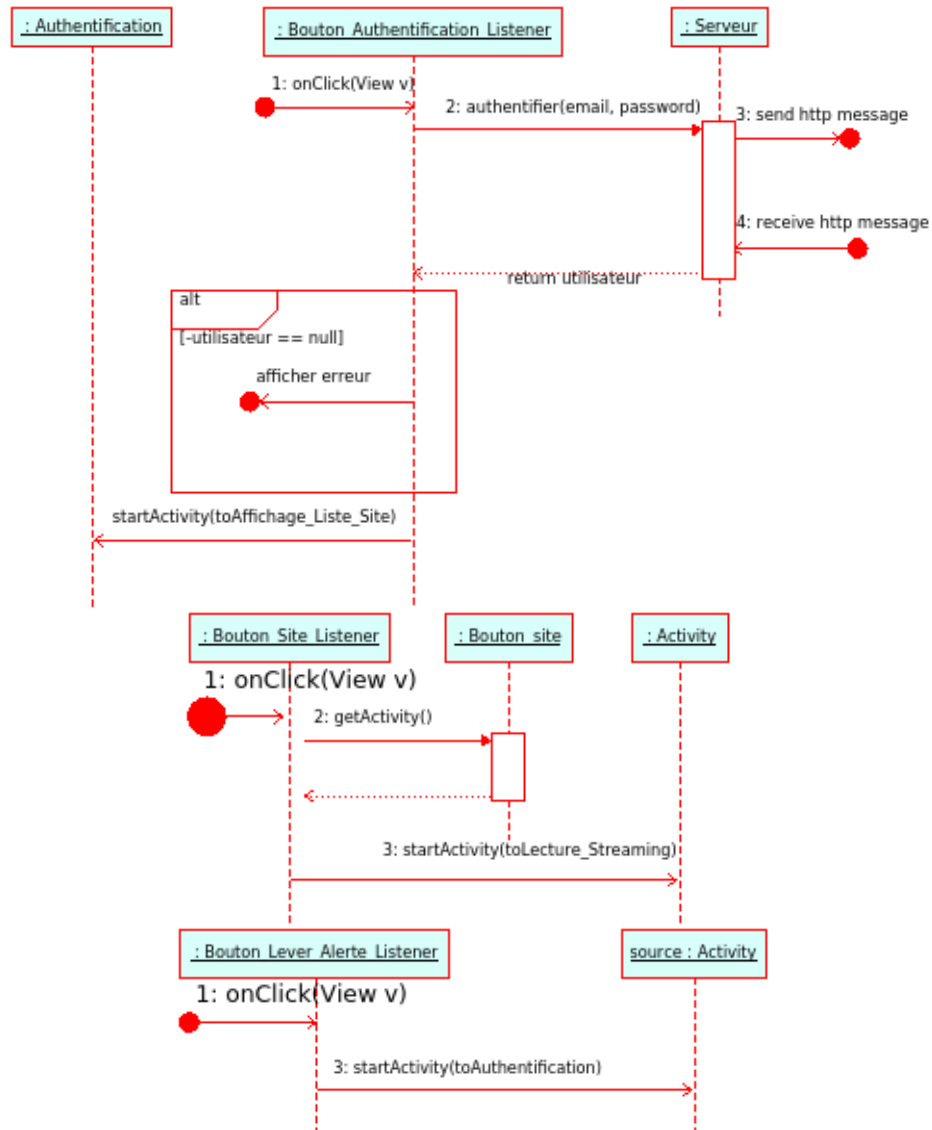


La classe *Serveur* est utilisée pour envoyer des messages à notre Serveur Rapace, sa structure est la suivante : (voir justification en annexe)



3.5 Diagramme de séquence

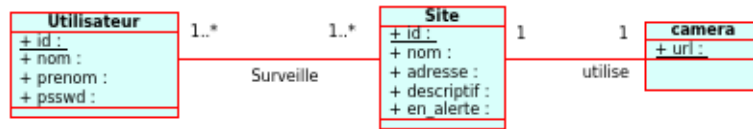
Nous montrons ici comment nos contrôleurs gèrent les transitions :



3.6 Base de données

Nous avons besoin d'une base de données afin d'authentifier un utilisateur, mémoriser des informations sur les utilisateurs, les sites, et les liens de surveillance. On décide qu'un utilisateur peut surveiller un nombre illimité de sites, et qu'un site peut être surveillé par un nombre illimité d'utilisateurs,

mais qu'un site ne dispose que d'une seule caméra (champ url). Il vient le schéma entité-association suivant :



Soient les relations :

Utilisateur (id, nom, prenom, psswd, email)
Site (id, nom, adresse, descriptif, url, en_alerte)
Surveille (id_utilisateur, id_site)

4 Réalisation et implémentation

Cette section présente la réalisation des fonctionnalités les plus importantes.

4.1 Interface Homme Machine :

Les Activités sont des classes spéciales auxquelles est associé un fichier XML définissant une interface statique. On peut aussi définir l'IHM en utilisant du code Java.

Si d'emblée cette pratique semble moins agréable qu'une définition par XML, elle offre en revanche la possibilité de modifier de notre interface à la volée. C'est ce que l'on appelle une interface dynamique.

L'Activité "*Affichage_Liste_Site*" liste l'ensemble des sites placés sous surveillance. Ce nombre étant variable, il est indispensable d'utiliser une interface dynamique.

Nous écrivons la partie statique dans le XML et la dynamique en Java, et reconstruisons cette partie chaque fois que l'activité revient au premier plan (callback : onResume()). C'est cette dynamisme qui permet de colorer un site en vert lorsque l'état d'alerte est faux (voir manuel en annexe), en rouge dans le cas contraire).

Pour écrire une interface dynamique, notre solution est de laisser un <LINEARLAYOUT> identifié que l'on manipulera dans le code Java.

```
<!-- Fichier XML -->
<LinearLayout android:id="@+id/content2" ... ></LinearLayout>

// Fichier Java
// Récupération Layout XML :
LinearLayout content =(LinearLayout) findViewById(R.id.content2);

// On le vide de son contenu :
content.removeAllViewsInLayout();

// Création d'une vue :
TextView no_site = new TextView(this);
no_site.setText("Vous ne surveillez actuellement aucun site.");
LinearLayout.LayoutParams layoutParam =
new LinearLayout.LayoutParams(/* paramètres layout */);
```

```
// Insertion d'une vue dans notre layout :  
content.addView(no_site, layoutParam);
```

(voir code source en annexe pour d'autres exemples.)

4.2 Communication client - serveur

Pour gérer l'authentification et déterminer les lieux qu'un utilisateur place sous surveillance, on utilise une base de données. Dans notre cas, il s'agit d'un serveur MySQL couplé à l'outil PHPMYADMIN ; ce sont des outils simples, et dont les performances suffisent largement. Voyons à présent comment client et serveur communiquent :

4.2.1 Côté client Rapace

Le package `RAPACE.NETWORK` régit l'ensemble des communications avec le serveur `APACHE`. La classe abstraite `MESSAGEHTTP` implémente les fonctionnalités indispensables à l'envoi de messages HTTP. Elle repose sur la classe `java.net.HTTPURLConnection`, recommandée par `developer.android` depuis la version 2.3. `MESSAGEPOST` spécialise `MESSAGEHTTP` pour la méthode `POST`.

La classe `MESSAGEHTTP` spécialise la classe `Java.lang.THREAD`. La communication avec le serveur Apache peut prendre un certain temps durant lequel l'application se fige. L'IHM n'est alors plus opérationnelle, et l'utilisateur peut ressentir un désagrément. Pire encore, le système Android peut décider de fermer l'application et afficher un message d'erreur si le client n'obtient pas rapidement une réponse !

La classe `Thread` définit ce que l'on appelle en français un **processus léger**.

Un processus léger n'est pas un processus au sens propre. (voir annexe)

On utilise la fonction `MESSAGE.START()` pour envoyer un message, et `THREAD.JOIN()` pour attendre la réponse.

```
MessagePost message = new MessagePost(url, parametres);  
message.start();  
message.join();
```

Quant à la classe `SERVEUR`, elle implémente plusieurs méthodes comme `authentifier()`, ou `lever_alerte()`, dont le rôle est d'envoyer un formulaire au bon script, et de traiter sa réponse.

4.2.2 Côté serveur Apache

Nous utilisons trois scripts : *authentication.php*, *demande_site_par_utilisateur.php*, *lever_alerte.php*, dont les rôles sont explicites. Les scripts sont formés de la façon suivante :

- 1 - On teste la présence des paramètres attendus :

```
if (!empty(trim($_POST['parametre1']))
    and !empty(trim($_POST['parametre2'])))
    ...
    and !empty(trim($_POST['parametre_n'])))
){ /* traitement */ }
```

- 2 - On ouvre une connexion avec PDO, utilisons une requête préparée, et renvoyons le résultat avec une boucle sous la forme :

```
$answer = $query->fetch(PDO::FETCH_ASSOC);
foreach ($answer as $value) {
    print $value . " "; }
```

(Voir annexe pour comprendre l'importance d'utiliser des requêtes préparées et autres questions de sécurité.)

Authentification : Le cryptage du mot de passe

Que se passerait-il si un individu parvenait à récupérer le contenu de notre base de données ? Il posséderait le mot de passe tout utilisateur, et avec, aurait accès à toutes les caméras enregistrées !

On évite ce genre de problème en cryptant la base de données. Le site php.net recommande l'utilisation du couple `PASSWORD_HASH()`, `PASSWORD_VERIFY()` pour crypter et vérifier si un mot de passe crypté et non-crypté correspondent. *authentication.php* procède en deux étapes : il charge les informations d'un utilisateur dont l'identifiant est donné, vérifie le mot de passe, et si les deux concordent, renvoie les informations au client.

```
$query = $bdd->prepare("SELECT * FROM Utilisateur WHERE email= :email");
$query->bindParam(':email', $email, PDO::PARAM_STR);
$query->execute();
$answer = $query->fetch(PDO::FETCH_ASSOC);
if ( password_verify($password, $answer['psswd']) ) {
    foreach ($answer as $value) {
        print $value . " "; } }
```

Détail : getSitesByUserId.php Ce script a pour mission de renvoyer l'ensemble des sites surveillés par un utilisateur dont on connaît l'identifiant.

```
if (!empty((int)htmlentities($_REQUEST['user_id']))) {
// Les id sont sous forme d'int, pas d'injection SQL possible.
$user_id = (int)$_REQUEST['user_id'];
$query = $bdd->prepare(
    "SELECT DISTINCT Site."
    "FROM Site " .
    "JOIN Surveillance " .
    "ON Site.id = Surveillance.id_site " .
    "WHERE Surveillance.id_utilisateur=" . $user_id
);
$query->execute();
```

La seule difficulté posée par ce script réside dans le renvoi des informations, que nous étudions en annexe.

Détail : lever_alerte.php

Nous appliquons les principes de construction énoncés précédemment en ajoutant une contrainte : pour lever une alerte, il faut re-authentifier l'utilisateur.

Cette mesure est faite pour éviter qu'un intrus déjoue le système d'alertes en envoyant une simple requête au serveur Rapace.

```
$statement = $bdd->prepare('
    UPDATE Site, Utilisateur, Surveillance
    SET en_alerte = 0
    WHERE Site.id = :id_site
    AND Utilisateur.email = :email
    AND Utilisateur.pswd = :password
    AND Utilisateur.id = Surveillance.id_utilisateur
    AND Surveillance.id_site = Site.id;
');

$statement->bindParam(':email', $_POST['email'], PDO::PARAM_STR);
// Remarque : Le mot de passe reçu correspond au mot de passe CRYPTÉ !
// (gardé en mémoire dans l'application cliente le temps de l'exécution)
// on peut donc le passer tel quel en parametre.
$statement->bindParam(':password', $_POST['password'], PDO::PARAM_STR);
```

```
$statement->bindParam(':id_site', $_POST['id_site'], PDO::PARAM_INT);

$statement->execute();
```

4.3 Mécanisme d'alerte :

Le serveur contacte l'utilisateur par sms. Après réception du message, l'application revient au premier plan en affichant le message d'alerte. Android permet aux développeurs d'interagir facilement avec les composants du téléphone. La classe SMSRECEIVER spécialise la classe BROADCASTRECEIVER qui permet d'écouter un composant Android. Lorsqu'un message arrive, il est accessible depuis les instances de BROADCASTRECEIVER. Dans notre cas, on récupère le message, on vérifie la provenance, et on invoque ALERTE si le numéro (pdu) correspond à celui du serveur.

```
public void onReceive(Context context, Intent intent) {
    // On récupère l'intent
    Bundle bundle = intent.getExtras();
    SmsMessage[] msgs;
    if (bundle != null) {
        Object[] pdus = (Object[]) bundle.get("pdus");
        msgs = new SmsMessage [pdus.length];

        /* Pour tout messages entrant */
        for (int i=0; i < msgs.length; i++) {
            SmsMessage message_entrant = SmsMessage.createFromPdu((byte[]) pdus[i]);
            /* Si le numéro correspond à celui du serveur (000) */
            if (message_entrant.getOriginatingAddress().equals("000")) {
                msgs[i] = message_entrant;

                /* On appelle l'Activité Alerte */
                Intent mainActivityIntent = new Intent(context, Alert.class);
                mainActivityIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
                context.startActivity(mainActivityIntent);
            }
        }
    }
}
```

4.4 Lecture streaming :

Différents types de streaming :

Il existe deux types de streaming :

- **Lecture progressive** : On lit un fichier (fini), stocké sur un serveur. On parle de lecture progressive lorsque l'application télécharge et lit le fichier *simultanément*. Le protocole HTTP supporte ce type de streaming. C'est le streaming utilisé par Youtube, par exemple.
- **Lecture continue** : On lit un flux vidéo. Le protocole HTTP ne supporte pas très bien ce mode de streaming, on utilise plutôt RTSP ou RTMP. C'est ce type de streaming qui est utilisé pour diffuser le flux d'une caméra.

L'API Android propose de lire une vidéo en streaming (lecture continue ou progressive) en utilisant un **VideoView** auquel on renseigne une *uri*.
Par exemple :

```
// On récupère la vue :  
VideoView video = (VideoView)findViewById(R.id.video);  
// On renseigne l'uri de la vidéo à lire :  
video.setVideoURI(Uri.parse(url));  
// Let's play :  
video.start();
```

5 Intérêts de ce projet

5.1 L'enrichissement personnel

Avant d'être une expérience informatique, ce projet est une expérience humaine. Les étudiants doivent coopérer pour mener à bien la réalisation de l'application. Cela nous a permis d'apprendre à communiquer, et à échanger pour clarifier parfois certains concepts que nous ne maîtrisions pas. Découvrir une nouvelle façon de programmer, ainsi que la richesse de l'univers Android, en dehors des cours magistraux est également une expérience intéressante. Dans le monde de l'entreprise nous serons certainement amenés à nous auto-former, et des projets comme celui-ci nous aident à acquérir la discipline et la maturité requise pour cela.

5.2 Des compétences techniques

Les notions abordées par ce projet sont nombreuses, et en font selon nous un très bon sujet pour l'apprentissage. Le code natif étant en JAVA, nous avons été amenés à renforcer notre maîtrise du langage.

Puisque la télésurveillance repose grandement sur les réseaux informatiques, il n'est pas étonnant de constater que nous avons approfondi nos connaissances dans ce domaine (l'architecture client-serveur et le protocole HTTP). Nous nous sommes également initiés aux problématiques du streaming et aux différents protocoles associés, comme aux formats d'encodage des images. Le projet revêt un côté système, puisqu'il a fallu (brièvement) étudier le fonctionnement du système Android afin d'implémenter un mécanisme d'alerte, et le multi-threading.

De plus, conjointement avec nos cours d'architecture du web et de base de données, ce projet nous a permis d'apprendre dans ces deux disciplines puisque nous avons dû écrire des scripts php régissant les interactions avec notre base de données. Enfin nous avons acquis certaines connaissances sur des outils comme PhpMyAdmin, les serveurs Apache et MySQL, le gestionnaire de versions SVN, le lecteur multi-média vlc dont nous n'avons pas parlé dans ce rapport mais qui nous a guidé tout au long de nos tests sur le streaming!

6 Conclusion

Nous avons, dans le présent document, proposé une solution simple pour répondre aux problèmes logiciels que pose la télésurveillance. La modularité du code et le respect des concepts objets permettront aux curieux de modifier et d'améliorer notre application avec aisance.

En effet les améliorations possibles sont nombreuses. Nous pouvons imaginer de nouvelles fonctionnalités venant enrichir notre application Rapace. Gérer la détection de mouvement, par exemple, de manière à compléter notre mécanisme d'alerte, où permettre à l'utilisateur d'enregistrer des vidéos à partir d'un flux, de conserver des clichés en cas d'intrusion, sont autant d'options pouvant améliorer l'expérience de l'utilisateur.

7 Bibliographie

**) Livres*

Programmer en Java de Claude Delannoy

Coder proprement, Martin, Robert C.

Cours Développement d'applications Mobiles sous Android , Abdelhak Djamel Seriali , Université Montpellier

**) Sites web*

[https ://www.java.com](https://www.java.com)

[https ://www.php.net.com](https://www.php.net.com)

[https ://www.openclassrooms.com](https://www.openclassrooms.com)

[http ://developer.android.com/](http://developer.android.com/)

[http ://www.developpez.com/](http://www.developpez.com/)

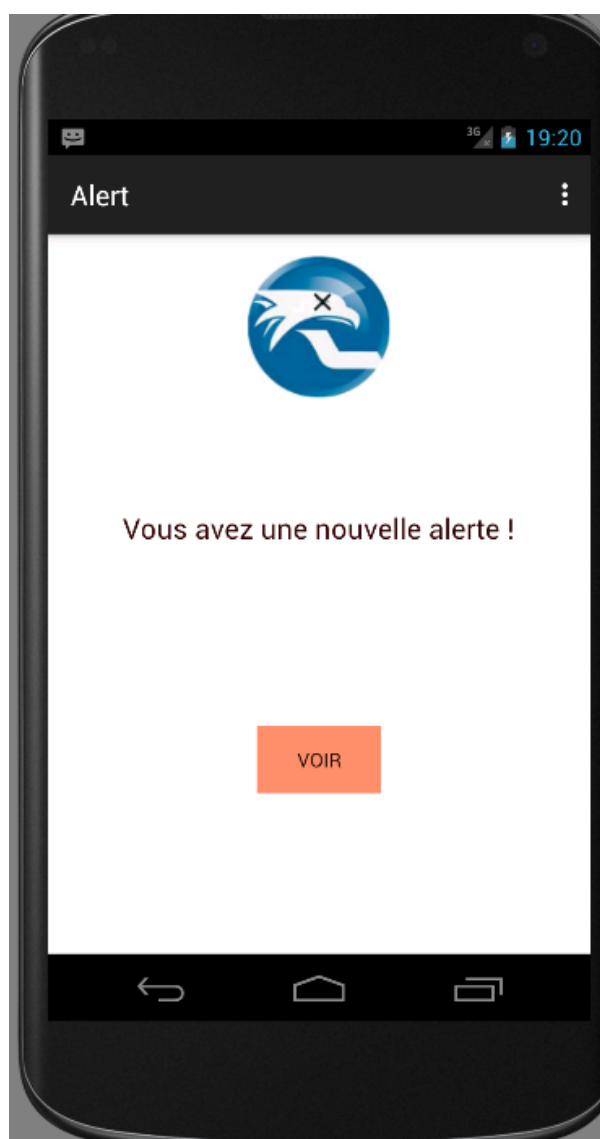
[http ://fr.wikipedia.org/wiki/Vidésurveillance](http://fr.wikipedia.org/wiki/Vidésurveillance)

8 Annexe

8.1 Mode d'emploi

Nous allons présenter une utilisation classique de notre application à travers la simulation d'une intrusion. On suppose que l'application est installée sur le mobile, et que l'utilisateur est enregistré dans la base de données.

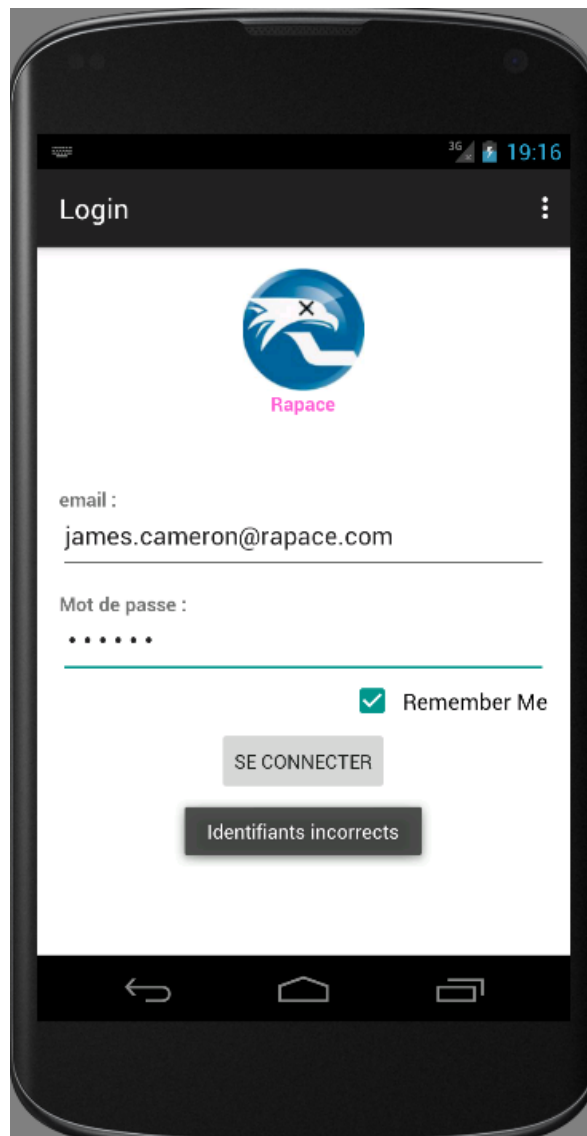
8.1.1 Réception de l'alerte



La pression du bouton "voir" nous redirige vers l'authentification.

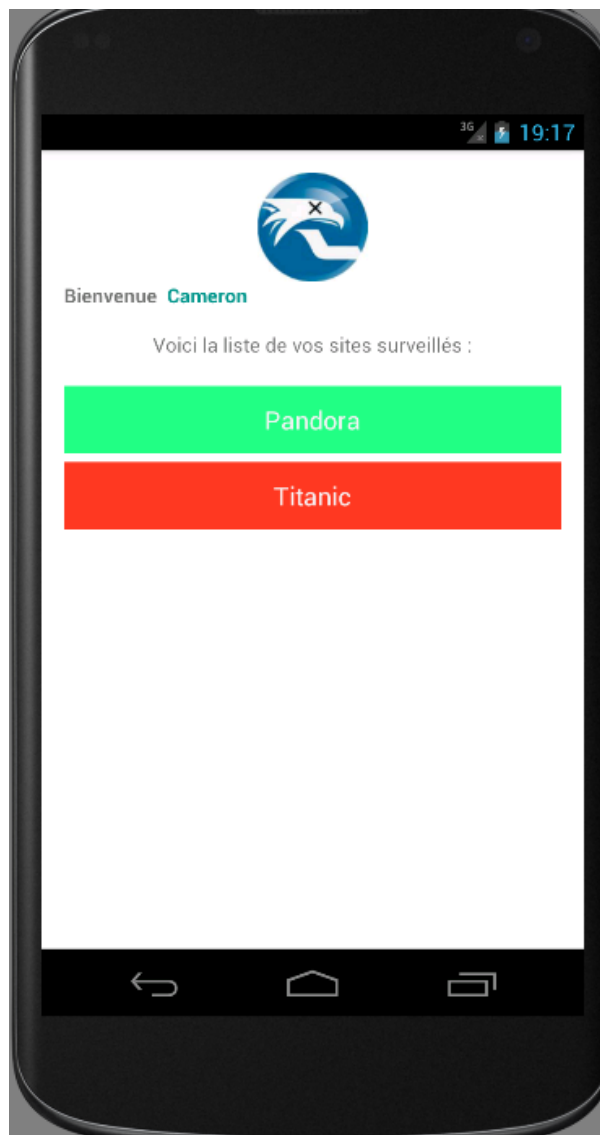
8.1.2 L'Authentification

Il s'agit alors de renseigner notre adresse email et notre mot de passe.



La pression du bouton "s'authentifier" nous redirige vers l'affichage des sites.

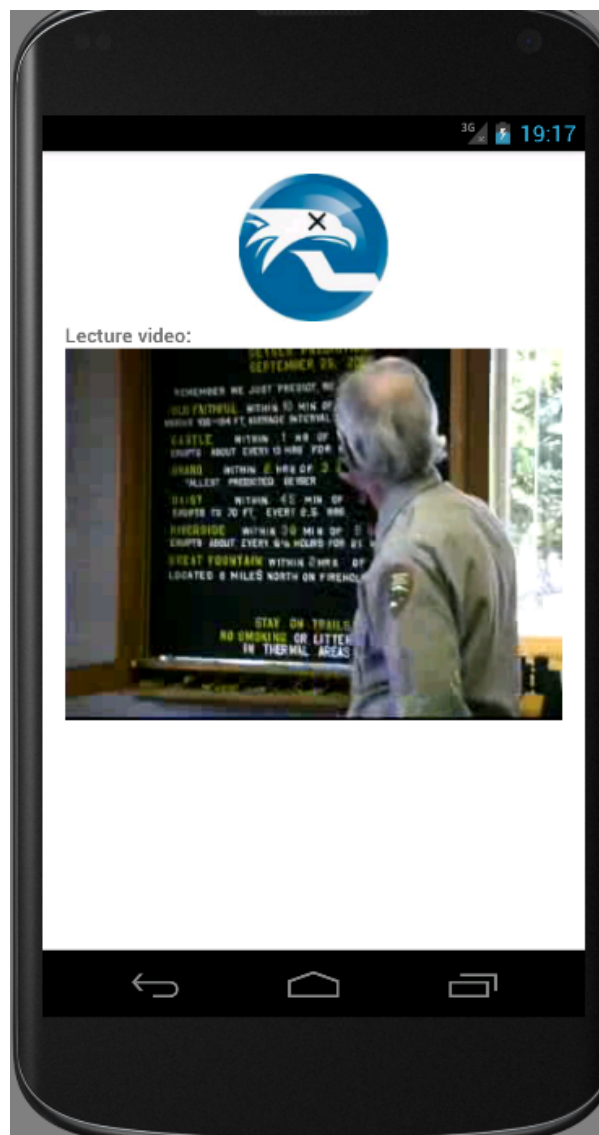
8.1.3 Affichage des sites



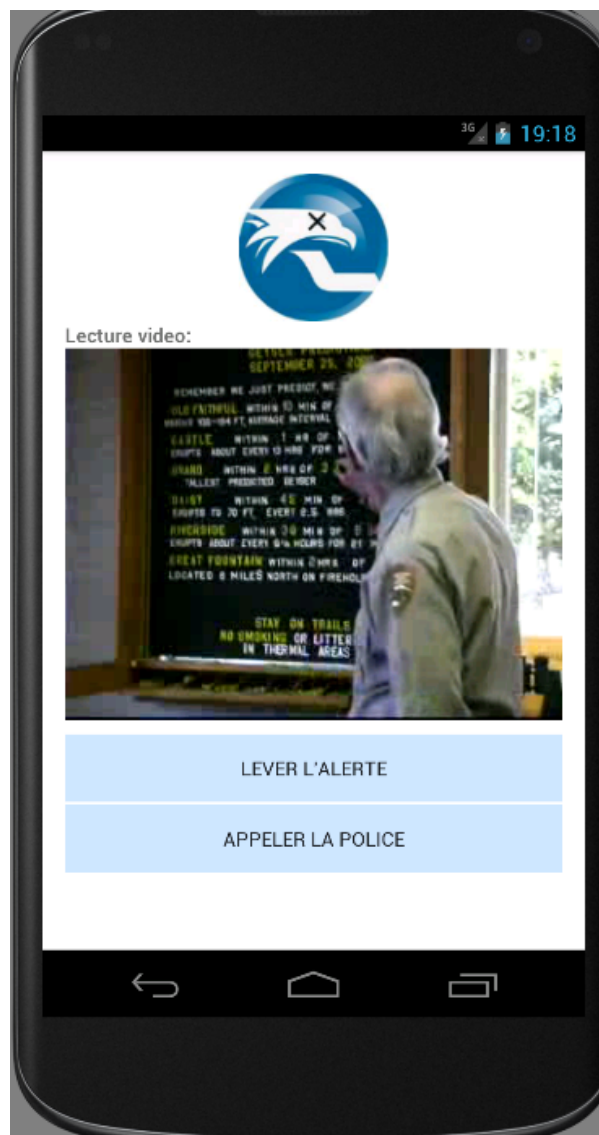
Les sites affichés en vert sont des sites pour lesquels aucune alerte n'est enregistrée, les sites rouge en revanche sont en état d'alerte !
Une pression sur un bouton vous redirigera vers les images du site.

8.1.4 Streaming

Suivant le site que vous avez choisi, l'affichage diffère :

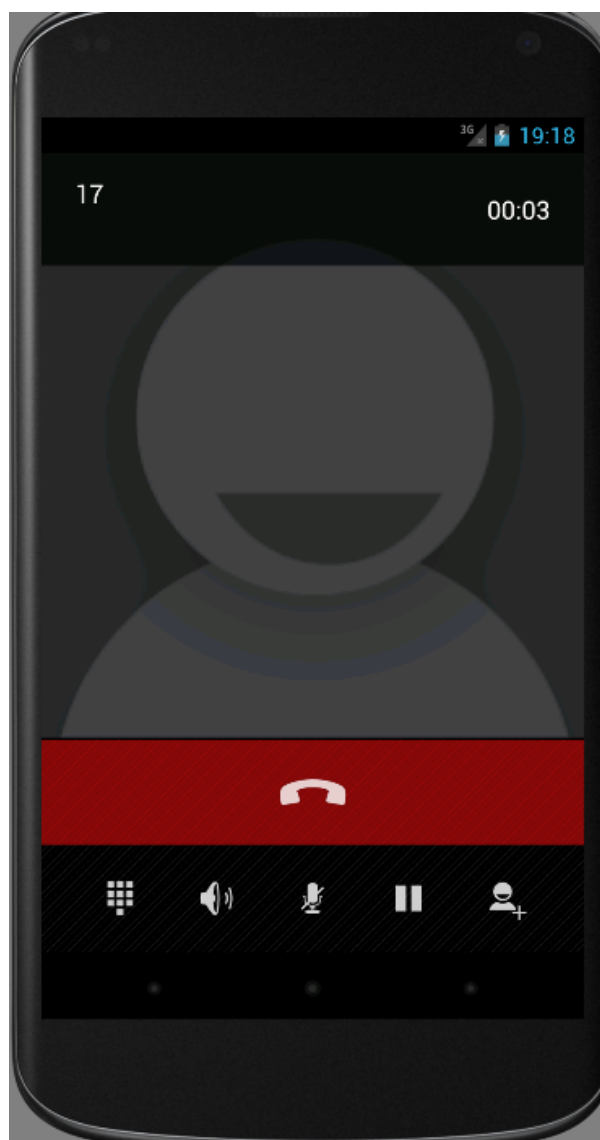


On visionne un site sans alerte.

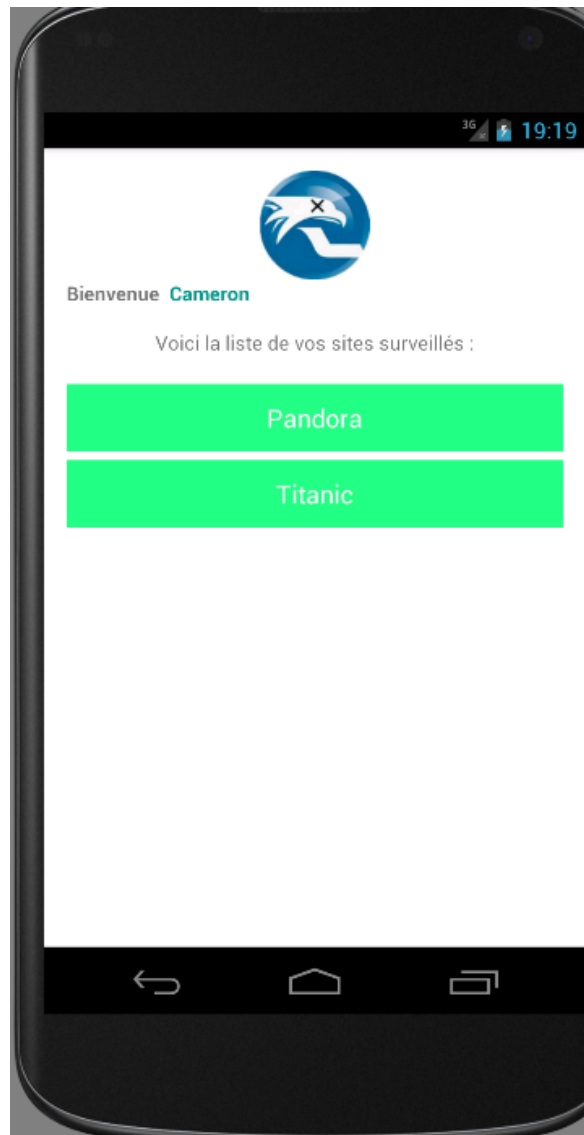


On visionne un site en alerte.

La pression du bouton "appeler la police" provoque l'appel de la police :



La pression du bouton "lever alerte" provoque la levée de l'alerte. Ainsi si l'on retourne en arrière nous obtenons l'affichage suivant :



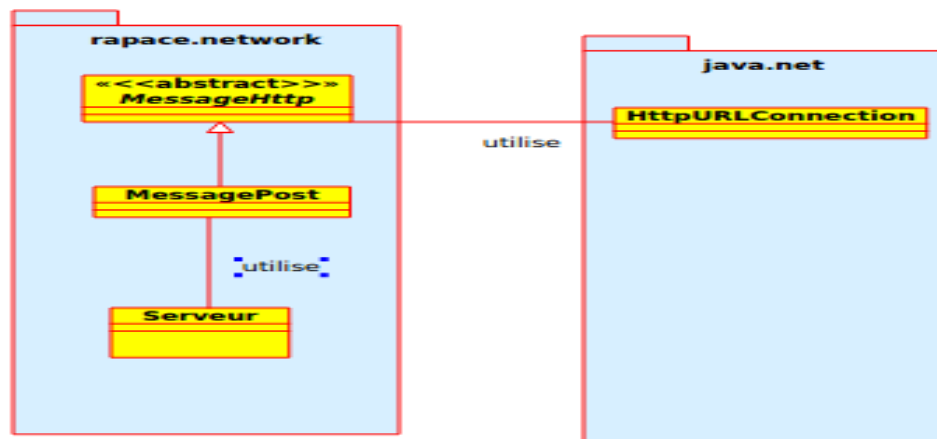
8.2 Complément sur les communications Client-Serveur

Nous reprenons dans cette section certaines explications déjà données dans le rapport que nous détaillons.

8.2.1 Pourquoi le package network ?

Le package `RAPACE.NETWORK` régit l'ensemble des communications avec le serveur `APACHE`.

Package rapace.network : Structure



La classe abstraite MESSAGEHTTP implémente les fonctionnalités indispensables à l'envoi de message HTTP, indépendamment la méthode utilisée. Elle repose principalement sur la classe `java.net.HttpURLConnection`, recommandée par `developer.android` depuis la version 2.3. D'autres classes permettent l'envoi de requêtes HTTP, mais des problèmes de compatibilité apparaissent suivant la version du mobile. MESSAGEPOST spécialise MESSAGEHTTP afin de prendre en charge la méthode POST.

Nous aurions pu créer une classe MESSAGEGET spécialisant MESSAGEHTTP pour la méthode GET, mais nous n'avons pas eu besoin d'utiliser cette méthode dans notre projet.

HttpURLConnection, supporte à elle seule l'envoi de message HTTP, quelque soit la méthode. On peut alors se demander pourquoi avoir créé le package RAPACE.NETWORK. Le site `developer.android` nous donne un exemple de son utilisation :

```
//Création de l'URL :
URL url = new URL("http://www.android.com/");
//Ouverture de connexion
HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
try {
```

```

//Configuration de la connection :
urlConnection.setDoOutput(true);
urlConnection.setChunkedStreamingMode(0);
//Création du flux d'émission :
OutputStream out = new BufferedOutputStream(urlConnection.getOutputStream());
//écriture dans ce flux :
writeStream(out);
//Creation du flux de réception :
InputStream in = new BufferedInputStream(urlConnection.getInputStream());
//Lecture du flux :
readStream(in); }
finally {
    //fermeture de connection :
    urlConnection.disconnect(); }

```

Le code suivant pose plusieurs problèmes :

Ce code génère un grand nombre d'exceptions !

On en compte pas moins de huit dans son adaptation à notre client Rapace. Or l'implémentation ci-dessus ne permet pas d'identifier clairement les exceptions et de les traiter. Plusieurs méthodes, par exemple, jettent une `JAVANULLPOINTEREXCEPTION`.

Solution : Une solution vise à découper le bloc try en autant de sous-blocs try qu'il y a de méthodes "À RISQUE". Mais avec un nombre important de méthodes à risque, notre code devient rapidement illisible.

Ce code gère un grand nombre de fonctionnalités :

Ouverture de connexion, configuration de connexion, création de flux (émission/réception) ainsi que leurs entrées/sorties, plus fermeture de connexion. Or, il est généralement préférable de n'attribuer qu'un rôle à une méthode.

Solution : Une solution consiste à découper notre code en autant sous routines, qu'il y a de commentaires dans notre exemple.

En conjuguant nos deux solutions, on obtient la méthode principale de MESSAGEPOST :

```
public void run() {
    ouvrir_connection();
    emettre_requete();
    setReponse_serveur(recevoir_reponse());
    fermer_connection(); }
```

Les sous-routines ci-dessus sont elles mêmes découpées en sous-routines. En plus d'améliorer le maintien de notre programme, la modularité permet aussi d'associer une sémantique à une sous-routine.

Package `rapace.network` : Relation entre les classes

Puisque `MESSAGEHTTP` réutilise les propriétés de `HTTPURLCONNECTION`, on pourrait se demander pourquoi avoir choisi d'associer les deux classes, alors qu'une relation de spécialisation semblait mieux indiquée pour illustrer les liens sémantiques entre nos classes.

Tout simplement parce que `MESSAGEHTTP` spécialise déjà la classe `Java.lang.Thread`. En effet, la communication avec le serveur Apache peut prendre un certain temps durant lequel l'application se fige. L'IHM n'est alors plus opérationnelle, et l'utilisateur peut ressentir un désagrément. Pire encore, le système Android peut décider de fermer notre application et afficher un message d'erreur si le client n'obtient pas rapidement de réponse !

Quant à la classe `SERVEUR`, elle implémente plusieurs fonctions comme `authentifier()`, ou `lever_alerte()`, dont le rôle est simplement de renseigner l'url du script à appeler et formater les paramètres à transmettre à la classe `MessagePost`. Lorsque `message` passe à l'état terminé, la méthode lit la réponse renvoyée par le serveur `APACHE` et effectue un traitement en conséquence.

8.2.2 Les processus légers ou Thread

"Pour éviter toute ambiguïté, il est important de préciser qu'un thread n'est pas un processus. En effet, les processus vivent dans des espaces virtuels isolés alors que les threads sont des traitements qui vivent ensemble au sein d'un même processus." ([http ://alwin.developpez.com/tutorial/JavaThread/](http://alwin.developpez.com/tutorial/JavaThread/)) En Java, la classe `Thread` permet de définir ce que l'on appelle en français un **processus léger**. Un processus léger n'est pas un processus au sens propre, mais un objet (dans notre cas) instancié dans un programme exécuté dans un certain processus.

Processus et Thread permettent une exécution en parallèle (au moins par

illusion), mais deux processus ont leurs propres segments mémoire (pile, statique, dynamique), alors que les processus légers utilisent les mêmes segments, celui du processus père. Cela rend la communication entre processus légers particulièrement agréable.

Chacun possède en revanche sa propre pile d'appel, permettant dans notre cas de détacher l'IHM de communication avec le serveur APACHE. En java, la création d'un processus léger se fait comme n'importe quelle instantiation d'objet.

L'exécution du processus léger démarre avec la méthode `THREAD.START()` qui effectue le traitement défini dans la redéfinition de `THREAD.RUN()`.

Pour attendre la fin de la fonction `THREAD.START()`, on utilise `THREAD.JOIN()`.

L'envoi d'un message HTTP suivant la méthode POST se fait par :

```
MessagePost message = new MessagePost(url, parametres);
message.start();
message.join();
```

8.2.3 Accès à la base de données

L'importance d'utiliser PDO et ses requêtes préparées ?

L'API PDO propose un ensemble de fonctionnalités optimisées et disponibles indépendamment du système de gestion de base de données.

Son usage (fortement recommandée par le site php.net) permet une plus grande portabilité du code, mais pas seulement.

La faille par injection SQL :

Une faille connue consiste à envoyer du code SQL dans un formulaire pour manipuler notre base de données sans en avoir les droits. En plus d'optimiser nos requêtes, l'utilisation de requêtes préparées permet de se prémunir contre ce type d'attaque.

8.2.4 Comment est encodée la réponse du serveur ?

Un problème s'est posé lorsque nous avons voulu sérialiser un ensemble de *Site* (code client) à partir des informations renvoyées par le serveur.

Pour ce faire nous récupérons d'abord tous les sites sous surveillance d'un utilisateur :

```
if (!empty((int)htmlentities($_REQUEST['user_id']))) {
// Les id sont sous forme d'int, pas d'injection SQL possible,
```

```
//pas de requetes préparées.
$user_id = (int)$_REQUEST['user_id']; // Néanmoins nous voulons
$query = $bdd->prepare(
    "SELECT DISTINCT Site."
    "FROM Site " .
    "JOIN Surveillance " .
    "ON Site.id = Surveillance.id_site ".
    "WHERE Surveillance.id_utilisateur=" . $user_id
);
$query->execute();
```

À ce stade, la réponse est contenue dans notre objet PDOStatement *\$query*. Nous désirons dans l'application client pouvoir construire autant d'objets *Site* que d'entrées (rows) trouvées dans la base de données. C'est à dire, pour une solution utilisant *StringTokenizer* :

```
String reponse_serveur = demande_site_par_utilisateur.getReponse_serveur();
if (!reponse_serveur.isEmpty()) {
    sites = new Vector<Site>();
    reponse = new StringTokenizer(reponse_serveur);
    while (reponse.hasMoreTokens()) {
        String long_token = reponse.nextToken(TOKEN);
        sites.add(new Site(long_token));
    }
}
```

Le code précédent fait plusieurs choses, il extrait une sous-chaîne délimitée par TOKEN et demande à la classe *Site* de créer une série d'objets à partir de cette chaîne, jusqu'à ce qu'il n'y ait plus de TOKEN dans *reponse_serveur*. De même, *Site(String data)* doit créer une nouvelle instance à partir d'une chaîne de caractères.

```
StringTokenizer strtok = new StringTokenizer(data, " ");

String str = strtok.nextToken(ATTR_TOKEN);
id = Integer.parseInt(str);

str = strtok.nextToken(ATTR_TOKEN);
nom = str;

str = strtok.nextToken(ATTR_TOKEN);
```

```

adresse = str;

str = strtok.nextToken(ATTR_TOKEN);
descriptif = str;

str = strtok.nextToken(ATTR_TOKEN);
url = str;

str = strtok.nextToken(ATTR_TOKEN);
etat_alerte = str;

```

La difficulté consiste donc à déterminer la valeur à donner à nos tokens. Comme "<" et ">" sont des caractères inhabituels pour décrire un site, nous avons choisis de remplacer TOKEN par "<" et ATTR_TOKEN par ">" dans notre code, et d'échapper systématiquement ces caractères lors de toutes les insertions dans la table *Rapace.Site*. Nous renvoyons finalement notre réponse à l'application client de la manière suivante :

```

while ($answer = $query->fetch(PDO::FETCH_ASSOC)) {
    foreach ($answer as $value) {
        print $value . '>';
    }
    print '<';
}

```

8.3 Nativité d'une application

Que l'on développe sous Android, iOS ou Windows, il existe trois types d'applications ayant chacune leurs avantages et leurs défauts.

- **Les applications natives** : Elles sont intégralement développées en langage natif (Java et XML pour Android). Ces langages ont l'avantage d'offrir de meilleures performances, il est compilé (en partie) et le système est optimisé pour les supporter. De plus, ils donnent la possibilité d'interagir avec le système d'exploitation.
- **Les applications web** : Elles récupèrent le contenu d'une page web. Sous Android cela se fait via la balise ouvrante XML `<WEBVIEW>` prenant en attribut l'url de la page encodée avec les technologies web usuelles. L'avantage de ces applications est leur portabilité : il ne s'agit plus de

compatibilité entre systèmes d'exploitations mais entre navigateurs web. Là où une application native exige trois implémentations différentes pour être correctement représentée sur le marché (AppStore, AndroidMarket, Windows Phone Store), une seule implémentation suffit à une application web. Les performances sont certes moins élevées, mais il faut noter que la majorité des applications ne requièrent pas de hautes performances. Le vrai défaut de ces applications réside dans le fait que généralement, par sécurité, le système d'exploitation interdit aux scripts l'accès aux composants du système (caméra, accès sms, répertoire, etc).

- **Les application hybrides** : De manière assez évidente, elles acceptent du code natif et du code issu d'une page web. Il s'agit d'un compromis entre application native et application web, permettant de tirer judicieusement parti de chacune.

8.4 Envoyer un SMS à l'émulateur Android

Un vrai téléphone possède un numéro, et le serveur envoie un sms d'alerte à ce numéro. Dans notre cas il s'agit d'une machine virtuelle dépourvue de numéro. Pour simuler l'envoi d'un sms, nous devons donc suivre la procédure suivante :

- On utilise la commande `TELNET #SERVEUR #PORT` pour se connecter au port utilisé par la machine virtuelle ; dans notre cas cela donne : `TELNET LOCALHOST 5554`. Le numéro de port utilisé par votre machine virtuelle est indiqué dans le coin supérieur gauche de votre fenêtre contenant la machine virtuelle.
- On peut alors utiliser les commandes livrées avec l'émulateur android `SMS SEND #NUM_TELEPHONE #CONTENU_MESSAGE` Par exemple : `sms send 007 "Mon nom est Bond, James Bond"` se traduit par la réception chez notre machine virtuelle d'un message provenant du numéro "007" contenant "Mon nom est Bond, James Bond".

8.5 Code source

Dans cette section nous présentons une partie du code utilisé par notre application, nous occultons par souci de clarté des commentaires ou des méthodes peu utiles au lecteur (on pense en particuliers aux accesseurs/mutateurs), ainsi que les imports.

8.5.1 Utilisateur

```
package com.example.quentin.rapace_v1.metier;

/** Classe representant un utilisateur inscrit */
public class Utilisateur {
    /** identifiant dans la base de donnees */
    private int id = -1;

    /** nom de l'utilisateur */
    private String nom = "non renseigné";

    /** prenom de l'utilisateur */
    private String prenom = "non renseigné";

    /** mot de passe (crypte) de l'utilisateur */
    private String psswd = "non renseigné";

    /** adresse email de l'utilisateur */
    private String email = "non renseigné";

    public Utilisateur(){}

    /**
     * Constructeur
     * @param data Reponse formatee du serveur.
     */
    public Utilisateur(String data) {
        //On utilise une instance de StringTokenizer pour parser notre réponse :
        StringTokenizer strtok = null;
        try {
            strtok = new StringTokenizer(data, " ");
        } catch (NullPointerException e) {
            /* Normalement il n'est pas nécessaire de catcher une RuntimeException,
            mais écrire un Log et quitter la fonction ici peut-être utile
            (aide à comprendre en cas d'erreur!) */
            Log.w("Bad Data", "Impossible d'instancier Utilisateur à partir de null.");
        }
    }
}
```

```
    }

    id = Integer.parseInt(strtok.nextToken());
    nom = strtok.nextToken();
    prenom = strtok.nextToken();
    psswd = strtok.nextToken();
    email = strtok.nextToken();
}

/**
 * Rassemble les donnees de l'utilisateur dans une chaine de caracteres.
 * @return Descriptif des donnees de l'utilisateur.
 */
public String toString() {
    return "[id] " + getId() + "\n[nom/prenom] " + getNom() + " " + getPrenom() +
}
}
```

8.5.2 Site

```
package com.example.quentin.rapace_v1.metier;

/** Classe representant un site sous surveillance. */
public class Site {

    /** identifiant du site dans la base de donnee */
    private int id = -1;

    /** nom du site */
    private String nom = "non renseigne";

    /** adresse du site */
    private String adresse = "non renseigne";

    /** descriptif du site */
    private String descriptif = "non renseigne";

    /** url de la video du site */
    private String url = "non renseigne";
```

```
/** etat d'alerte du site */
private String etat_alerte = "non renseigne";

/**
 * Constructeur
 * @param data String contenant le réponse (formatee) du serveur.
 */
public Site(String data) {
    StringTokenizer strtok = null;
    try {
        strtok = new StringTokenizer(data, " ");
    } catch (NullPointerException e) {
        Log.e("Bad Data",
            "Impossible d'instancier Utilisateur à partir de null.");
    }

    String str = null;
    str = strtok.nextToken(">");
    id = Integer.parseInt(str);

    str = strtok.nextToken(">");
    nom = str;

    str = strtok.nextToken(">");
    adresse = str;

    str = strtok.nextToken(">");
    descriptif = str;

    str = strtok.nextToken(">");
    url = str;

    str = strtok.nextToken(">");
    etat_alerte = str;
}
}
```

8.5.3 MessageHTTP

```
/**
 * Classe abstraite implémentant des fonctions de bases nécessaires
 * à l'envoi de requête Http.
 */
public abstract class MessageHttp extends Thread {
    /** adresse de la cible */
    private URL url = null;

    /** paramètre de la requête */
    private String param = null;

    /** classe gérant l'envoi et la réception des requêtes Http */
    private HttpURLConnection connection = null;

    /** reponse obtenue apres traitement de notre requete */
    private static String reponse_serveur = null;

    /**
     * Constructeur
     * @param url adresse de la cible
     * @param param paramètre de la requête
     */
    public MessageHttp(String url, String param) {
        try {
            this.url = new URL(url);
        } catch (MalformedURLException e) {
            Log.e("Erreur", "URL invalide (malformée ?)");
        }
        this.param = param;
    }

    /**
     * Accesseur vers la reponse obtenu apres traitement de la requete.
     * Attention aux problemes de synchronisation.
     * @return reponse obtenue apres traitement de la requete
     */
    public String getReponse_serveur() {
        return reponse_serveur;
    }
}
```

```
/**
 * Mutateur vers la reponse obtenu apres traitement de la requete
 * @param reponse_serveur
 */
public void setReponse_serveur(String reponse_serveur) {
    this.reponse_serveur = reponse_serveur;
}

/**
 * SubClassResponsability
 */
@Override
public void run(){}

/**
 * Ouvre une nouvelle connection.
 */
public void ouvrir_connection() {
    try {
        connection = (HttpURLConnection) url.openConnection();
    } catch (IOException e) {
        Log.e("Erreur", "Impossible d'accéder à l'url demandée "
            + url.toString());
    }
}

/**
 * Ferme une connection active.
 */
public void fermer_connection() {
    connection.disconnect();
}

/**
 * Ouverture du tampon de reception
 * @return tampon de reception ouvert (si reussite)
 */
public BufferedReader obtenir_tampon_reception() {
```

```
BufferedReader _tampon_reception = null;
// Ouverture du flux :
try {
    _tampon_reception = new BufferedReader
        (new InputStreamReader(connection.getInputStream()));
} catch (IOException e1) {
    Log.e("Erreur",
        "Erreur dans la réception du flux en provenance de "
        + url.toString()); }
return _tampon_reception;
}

/**
 * Encadre la reception d'une reponse serveur
 * @return reponse a la requete
 */
public String recevoir_reponse() {
    BufferedReader tampon_reception = obtenir_tampon_reception();
    String message = lire(tampon_reception);
    // Fermeture du tampon :
    try {
        tampon_reception.close();
    } catch (IOException e1) {
        Log.e("Erreur",
            "Impossible de fermer le flux de réception en provenance de "
            + url.toString()); }
    return message;
}

/**
 * Lit le contenu d'un tampon
 * @param bufferedReader tampon a lire
 * @return String fidele au contenu du tampon
 */
public String lire(BufferedReader bufferedReader) {
    String line;
    StringBuffer sb = new StringBuffer();
    try {
        while ((line = bufferedReader.readLine()) != null) {
            sb.append(line);
        }
    }
```

```
    } catch (IOException e1) {
        Log.e("Erreur", "Erreur dans la lecture du flux en provenance de "
            + url.toString());
    }
    return sb.toString();
}
}
```

8.5.4 MessagePOST

```
package com.example.quentin.rapace_v1.network;

/**
 * Permet l'envoi de requete HTTP suivant la methode POST
 */
public class MessagePost extends MessageHttp {

    /**
     * Constructeur
     * @param url adresse de la cible
     * @param param parametres de la requete
     */
    public MessagePost(String url, String param) {
        super(url, param);
    }

    /**
     * Traitement effectue par le Thread a son execution
     */
    @Override
    public void run(){
        ouvrir_connection();
        emettre_requete();
        setReponse_serveur(recevoir_reponse());
        fermer_connection();
    }

    /**
     * Ouverture du tampon d'emission
     * @return tampon d'emission ouvert
     */
}
```

```
*/
public OutputStreamWriter obtenir_tampon_emission() {
    OutputStreamWriter _tampon_emission = null;
    try {
        _tampon_emission = new OutputStreamWriter(getConnection().
                                                    getOutputStream());
    } catch (IOException e) {
        Log.e("Erreur", "Impossible d'obtenir un flux de sortie.");
    }
    return _tampon_emission;
}

/**
 * Preparation de la requete : chargement des parametres en tampon.
 * @param _tampon_emission tampon d'emission ouvert
 */
public void preparerRequete(OutputStreamWriter _tampon_emission) {
    // On y écrit notre morceau de requête HTTP contenu dans param :
    try {
        _tampon_emission.write(getParam());
    } catch (IOException e) {
        Log.e("Erreur", "Impossible d'écrire sur le flux sortant.");
    }
}

/**
 * Envoi une requete HTTP suivant la methode POST
 * @param _tampon_emission tampon d'emission ouvert et prepare
 */
public void envoyerRequete(OutputStreamWriter _tampon_emission) {
    // On vide le buffer afin d'envoyer notre requête HTTP :
    try {
        _tampon_emission.flush();
    } catch (IOException e) {
        Log.e("Erreur", "Impossible d'envoyer la requete HTTP.");
    }
}

/**
 * Met en forme et emet une requete
 */
```

```
public void emettre_requete() {
    // On autorise un tampon pour l'écriture de la requête :
    getConnection().setDoOutput(true);
    getConnection().setChunkedStreamingMode(0);

    OutputStreamWriter tampon_sortie = obtenir_tampon_emission();
    if (tampon_sortie == null) {
        Log.e("tampon" , " nul");
    }
    preparerRequete(tampon_sortie);
    envoyerRequete(tampon_sortie);

    // On ferme le tampon car il ne nous sera plus utile :
    try {
        tampon_sortie.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
```

8.5.5 Serveur

```
package com.example.quentin.rapace_v1.network;

/** Assure la liaison entre l'application et le serveur Rapace */
public class Serveur {
    /** url du script d'authentification */
    private static final String url_script_authentification =
        "http://10.0.2.2/~quentin/authentification.php";
    // 10.0.2.2 = 127.0.0.1 pour android

    /** url du script listant les sites que surveille un utilisateur
        a partir de son identifiant */
    private static final String url_script_demander_site_par_utilisateur = ...
    /** url du script permettant de lever une alerte */
    private static final String url_script_lever_alerte = ...

    /**
     * Envoie une requete permettant d'authentifier un utilisateur
     */
}
```

```
* enregistre selon son email et son mot de passe.
* @param email adresse email du client
* @param password mot de passe du client (non crypte)
* @return Une instance d'Utilisateur
*/
public static Utilisateur authentifier(String email, String password) {
    String identifiants = preparer_identifiants(email, password);

    MessagePost demande_authentification = new MessagePost(
        url_script_authentification, identifiants);
    demande_authentification.start();
    attendre_fin(demande_authentification);

    String reponse = demande_authentification.getReponse_serveur();
    /* "String reponse" contient une chaine contenant l'ensemble des
    attributs d'un unique utilisateur ou est nulle. */

    Utilisateur utilisateur = null;
    if (!reponse.isEmpty()) {
        utilisateur = new Utilisateur(reponse);
    }
    return utilisateur;
}

/**
 * Sous routine s'occupant de la preparation des arguments de la
 * requete d'authentification
 * @param email adresse email du client
 * @param password mot de passe du client (non crypte)
 * @return String formatee exprimant les parametres de notre requete.
 */
private static String preparer_identifiants
    (String email, String password) {
    String identifiants = null;
    try {
        identifiants = URLEncoder.encode("email", "UTF-8")
            + "=" + URLEncoder.encode(email, "UTF-8")
            + "&" + URLEncoder.encode("password", "UTF-8")
            + "=" + URLEncoder.encode(password, "UTF-8");
    } catch (UnsupportedEncodingException e) {
        Log.e("Error", "Argument mal formé ?");
    }
}
```

```
        e.printStackTrace();
    }
    return identifiants;
}

/**
 * Envoie une requete permettant de lister les sites surveilles
 * par un utilisateur.
 * @param id_utilisateur numero identifiant un utilisateur enregistre
 * @return Liste des sites sous la surveillance de l'utilisateur renseigne
 */
public static Vector<Site> charger_site(String id_utilisateur) {
    Vector<Site> sites = null;
    String id = preparer_requete_site(id_utilisateur);
    StringTokenizer reponse = null;

    MessagePost demande_site_par_utilisateur =
        new MessagePost(url_script_demander_site_par_utilisateur, id);
    demande_site_par_utilisateur.start();
    attendre_fin(demande_site_par_utilisateur);

    String reponse_serveur = demande_site_par_utilisateur.
        getReponse_serveur();
    if (!reponse_serveur.isEmpty()) {
        sites = new Vector<Site>();
        reponse = new StringTokenizer(reponse_serveur);
        while (reponse.hasMoreTokens()) {
            String long_token = reponse.nextToken("<");
            sites.add(new Site(long_token));
        }
    }
    return sites;
}

/**
 * Envoie d'une requete levant un etat d'alerte pour un site donne
 * @param email adresse email du client
 * @param password mot de passe du client (crypte)
 * @param id_site numero identifiant un utilisateur enregistre
 */
```

```
public static void lever_alerte(String email, String password,
                                String id_site) {
    String parametres = preparer_requete_alerte(email, password, id_site);
    MessagePost demande_levee = new MessagePost(url_script_lever_alerte,
                                                  parametres);

    demande_levee.start();
    attendre_fin(demande_levee);
}

/**
 * Attendre qu'une requete HTTP obtienne reponse
 * @param messageHttp requete dont on souhaite attendre la reponse
 */
private static void attendre_fin(MessageHttp messageHttp) {
    try {
        messageHttp.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}
```

8.5.6 SMSReceiver

```
package com.example.quentin.rapace_v1.service;

/**
 * Classe s'occupant de la détection des alertes.
 */
public class SMSReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle bundle = intent.getExtras();
        SmsMessage[] msgs;
        if (bundle != null) {
            /* Les pdus correspondent aux numéros de téléphone des expéditeurs */
            Object[] pdus = (Object[]) bundle.get("pdus");
            msgs = new SmsMessage [pdus.length];
            /* Pour tout messages entrant */
        }
    }
}
```

```
    for (int i=0; i < msgs.length; i++) {
        SmsMessage message_entrant =
            SmsMessage.createFromPdu((byte[]) pdus[i]);
        /* Si le numéro correspond à celui du serveur (000) */
        if (message_entrant.getOriginatingAddress().equals("000")) {
            msgs[i] = message_entrant;
            /* On appelle l'Activité Alerte */
            Intent mainActivityIntent = new Intent(context, Alert.class);
            mainActivityIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            context.startActivity(mainActivityIntent);

            Intent broadcastIntent = new Intent();
            broadcastIntent.setAction("SMS_RECEIVED_ACTION");
            context.sendBroadcast(broadcastIntent);
        }
    }
}
```